Lecture 12:
Transformers: Self-Attention Networks (contd.)

Instructor: Swabha Swayamdipta

USC CSCI 544 Applied NLP

Oct 03, Fall 2024

Some slides adapted from Dan Jurafsky and Chris Manning

USC Viterbi

Announcements

- Today: Quiz 3 **(requires lockdown browser!)**
- Tomorrow through 10/8: Please fill out a short survey feedback
- Next Tue:
  - HW2 due - please follow naming format etc. (see Brightspace announcement)
  - Guest lecture by TA Sayan Ghosh on PyTorch for Transformers
- Next Thu: No class / Fall Break
- Tue 10/15: Midterm Exam
  - 1 hr - format similar to quizzes
- HW1 / Project Proposal grades will be available by the end of the week
- Sign up sheet now open for Paper Presentation and Final Project Presentation dates (see Brightspace announcement)

Lecture Outline

- Recap: Transformers
- Quiz 3
- Transformers as Encoders, Decoders and Encoder-Decoders
- The pre-training and fine-tuning paradigm
  - Pre-training Decoder-Only Models
  - Pre-training Encoder-Only Models
  - Pre-training Encoder-Decoder Models

Recap: Transformers

Attention Variants

- In general, we have some keys $\mathbf{h}_1, \ldots, \mathbf{h}_N \in \mathbb{R}^{d_1}$ and a query $\mathbf{q} \in \mathbb{R}^{d_2}$
- Attention always involves
  1. Computing the attention scores, $e(\mathbf{q}, \mathbf{h}_{1:N}) \in \mathbb{R}^N$ → Can be done in multiple ways!
  2. Taking softmax to get attention distribution $\alpha_t = \text{softmax}(e(\mathbf{q}, \mathbf{h}_{1:N})) \in [0,1]^N$
  3. Using attention distribution to take weighted sum of values:

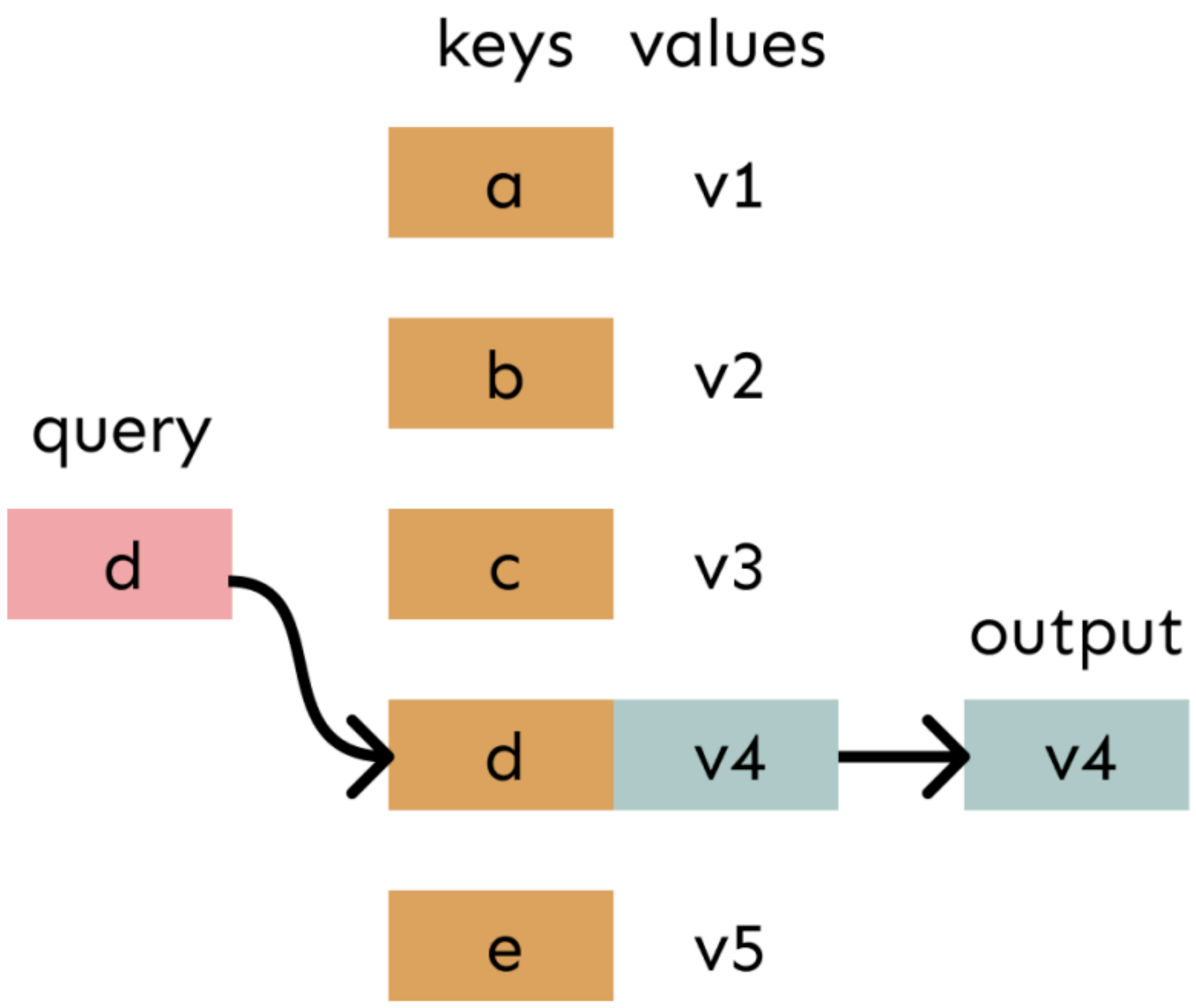
$$\mathbf{c}_t^{att} = \sum_{i=1}^{N} \alpha_{t,i} \mathbf{h}_i \in \mathbb{R}^{d_1}$$

This leads to the attention output $\mathbf{c}_t^{att}$ (sometimes called the attention context vector)

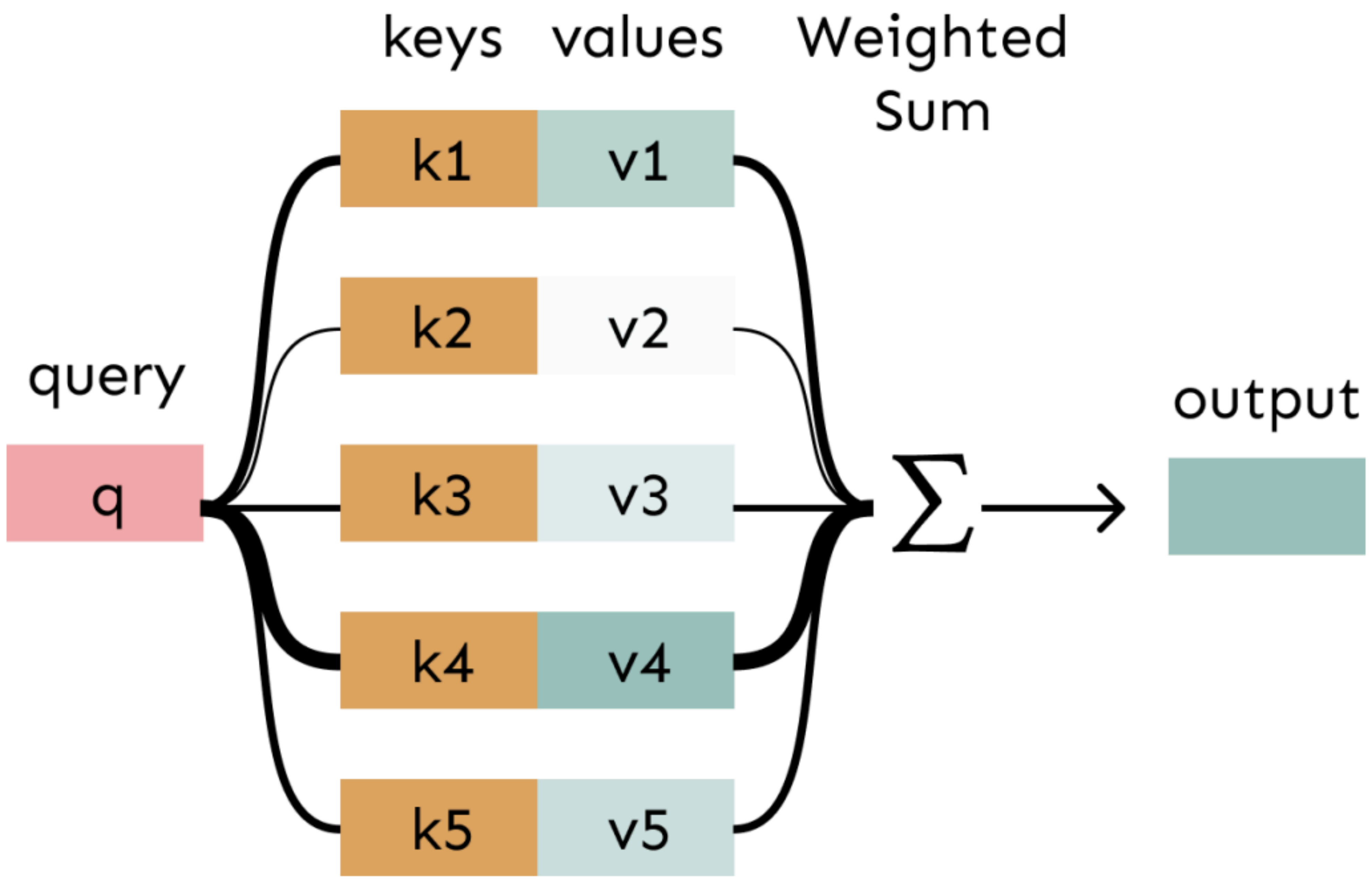
Attention and lookup tables

Attention performs fuzzy lookup in a key-value store

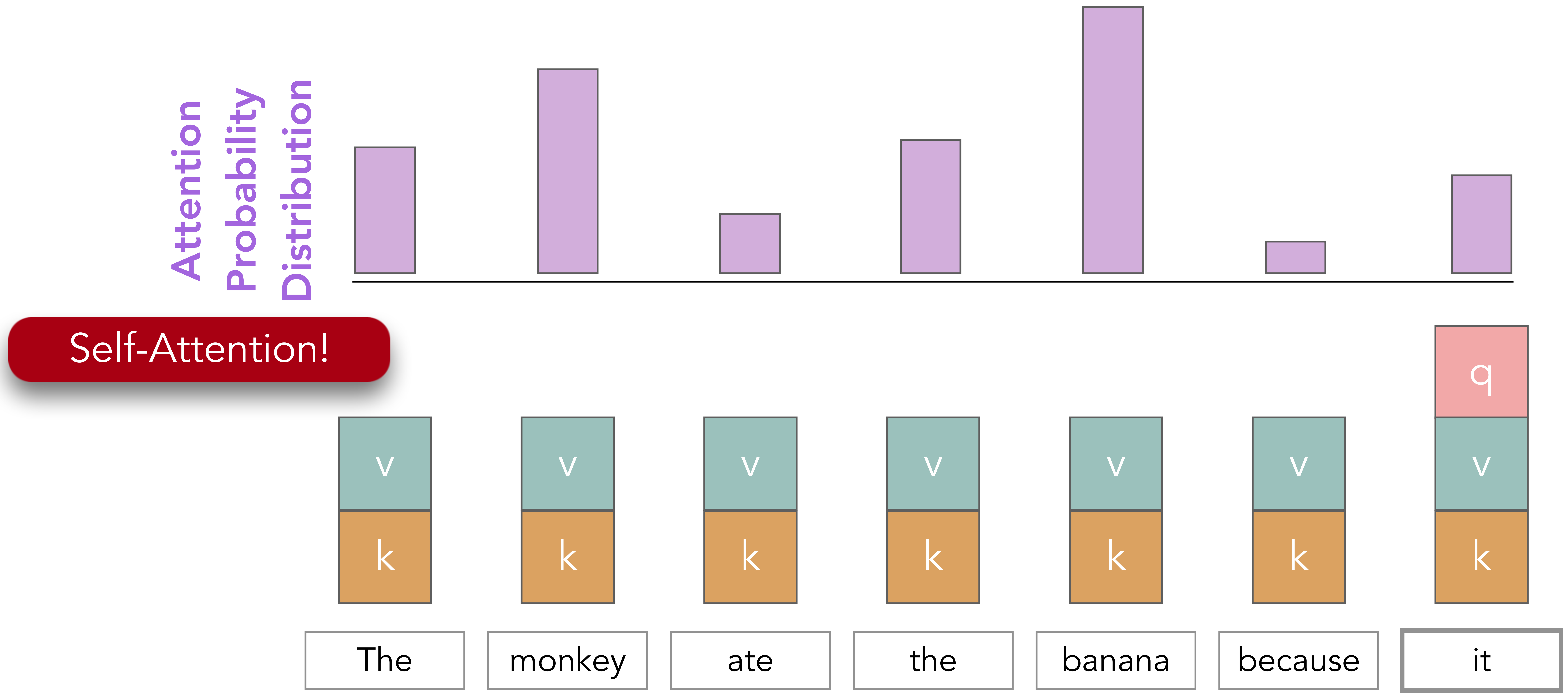
In a lookup table, we have a table of keys that map to values. The query matches one of the keys, returning its value.



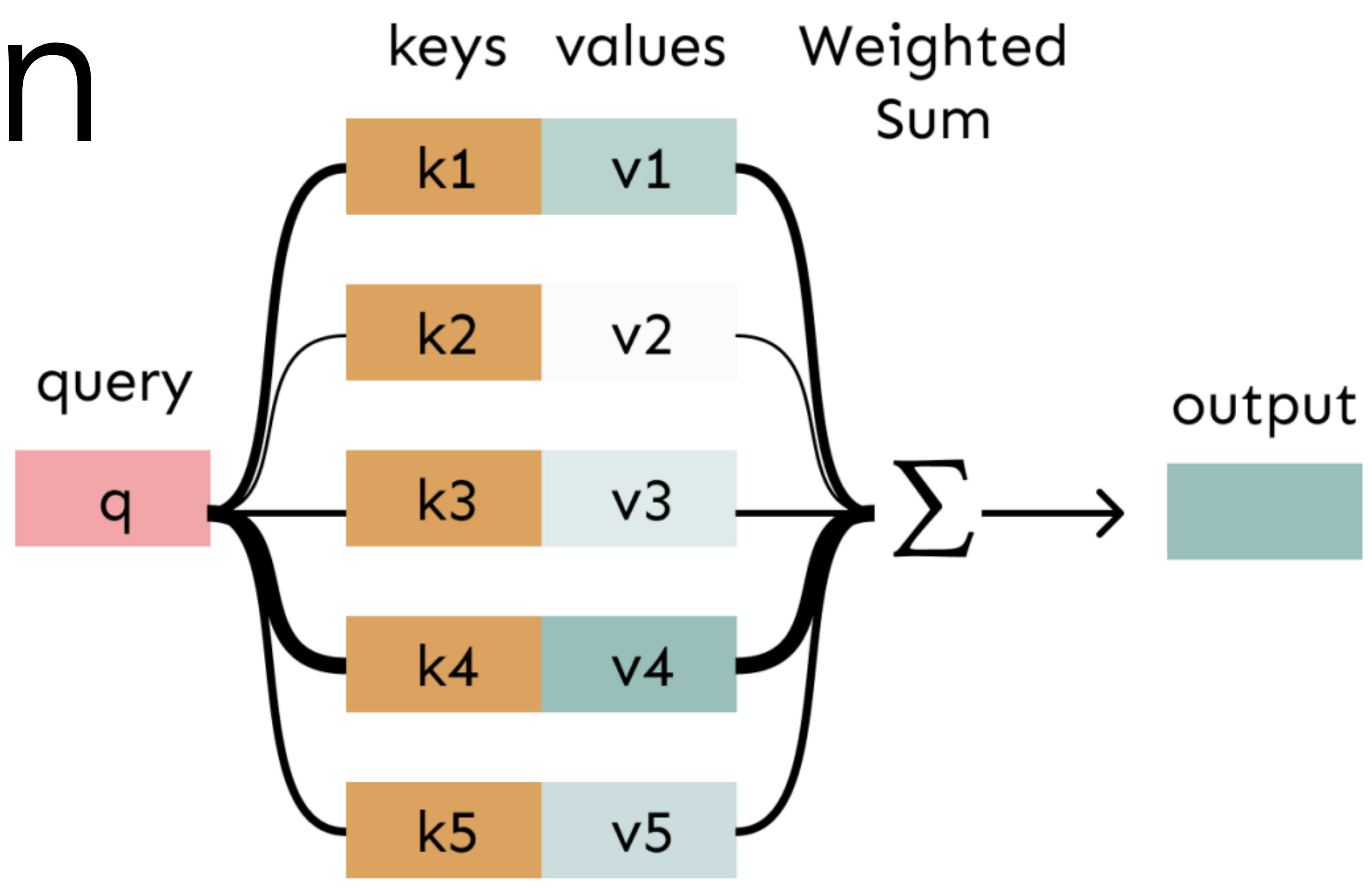
In attention, the query matches all keys softly, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



Self-Attention: Attention in the decoder



Self-Attention



Keys, Queries, Values from the same sequence

Let $w_{1:N}$ be a sequence of words in vocabulary V

For each w_i , let $\mathbf{x}_i = \mathbf{E}_{w_i}$, where $\mathbf{E} \in \mathbb{R}^{d \times V}$ is an embedding matrix.

1. Transform each word embedding with weight matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, each in $\mathbb{R}^{d \times d}$

$q_i = Q\boldsymbol{x}_i$ (queries) $\quad k_i = K\boldsymbol{x}_i$ (keys) $\quad v_i = V\boldsymbol{x}_i$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

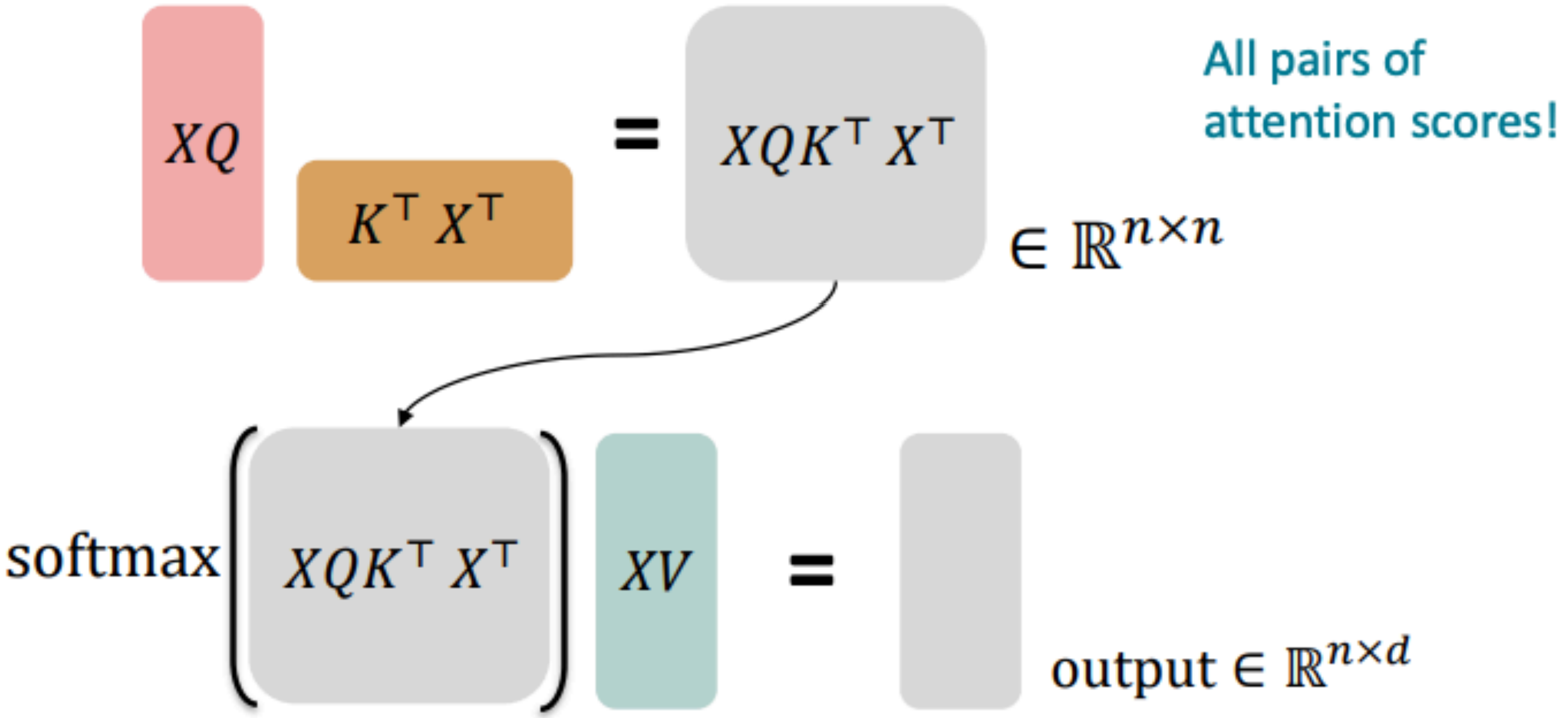
3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij}\, \boldsymbol{v}_i$$

Self-Attention as Matrix Multiplications

- Key-query-value attention is typically computed as matrices.
  - Let $\mathbf{X} = [\mathbf{x}_1; \ldots; \mathbf{x}_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors
  - First, note that $\mathbf{XK} \in \mathbb{R}^{n \times d}$, $\mathbf{XQ} \in \mathbb{R}^{n \times d}$, and $\mathbf{XV} \in \mathbb{R}^{n \times d}$
  - The output is defined as softmax$(\mathbf{XQ}(\mathbf{XK})^T)\mathbf{XV} \in \mathbb{R}^{n \times d}$

First, take the query-key dot products in one matrix multiplication: $\mathbf{XQ}(\mathbf{XK})^T$



Next, softmax, and compute the weighted average with another matrix multiplication.

Transformers are Self-Attention Networks

- Self-Attention is the key innovation behind Transformers!
- Transformers (self-attention networks) map sequences of input vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$ of the same length.
- Made up of stacks of Transformer blocks
  - each of which is a multilayer network made by combining
    - simple linear layers,
    - feedforward networks, and
    - self-attention layers
  - No more recurrent connections!

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

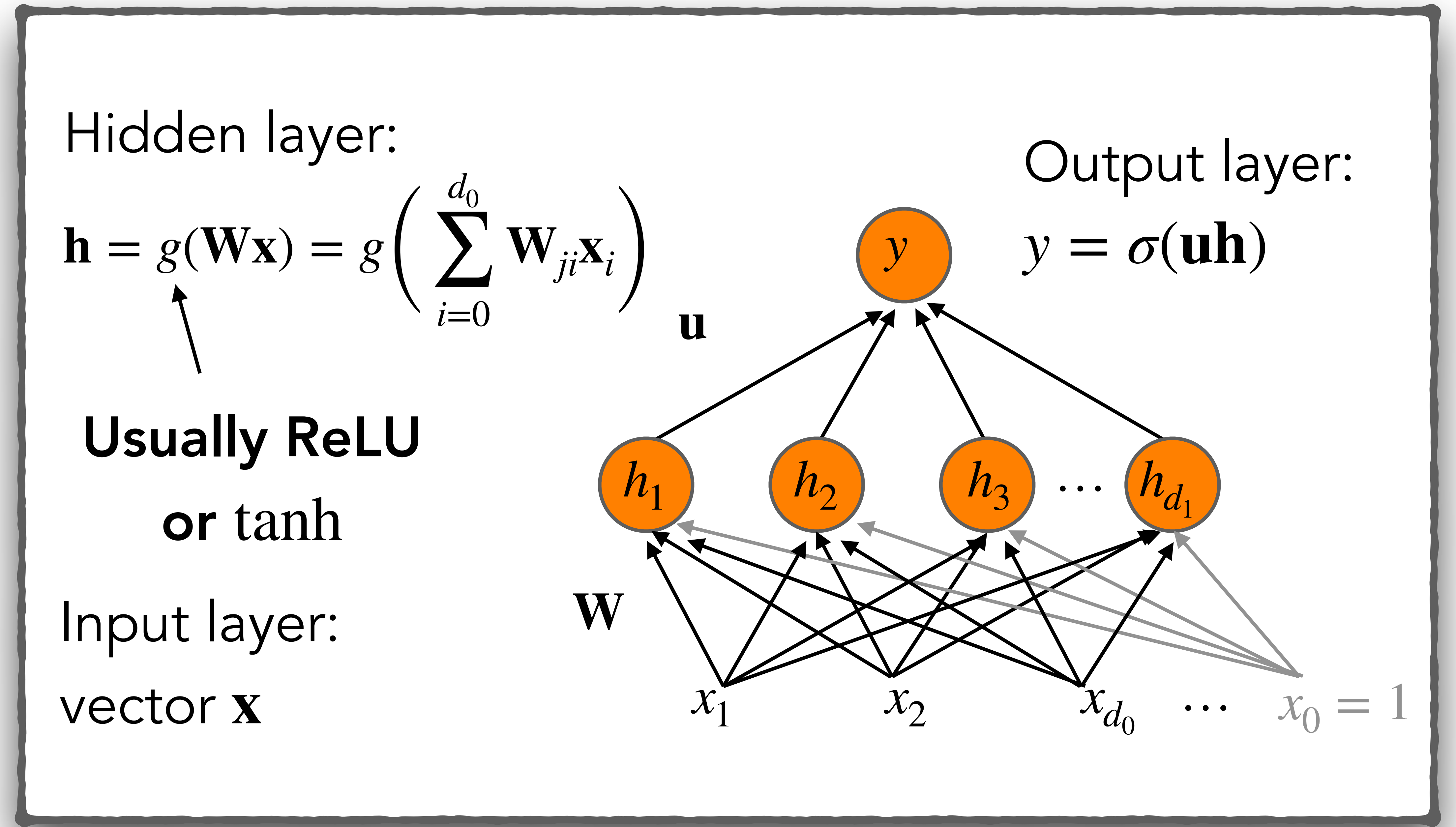
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

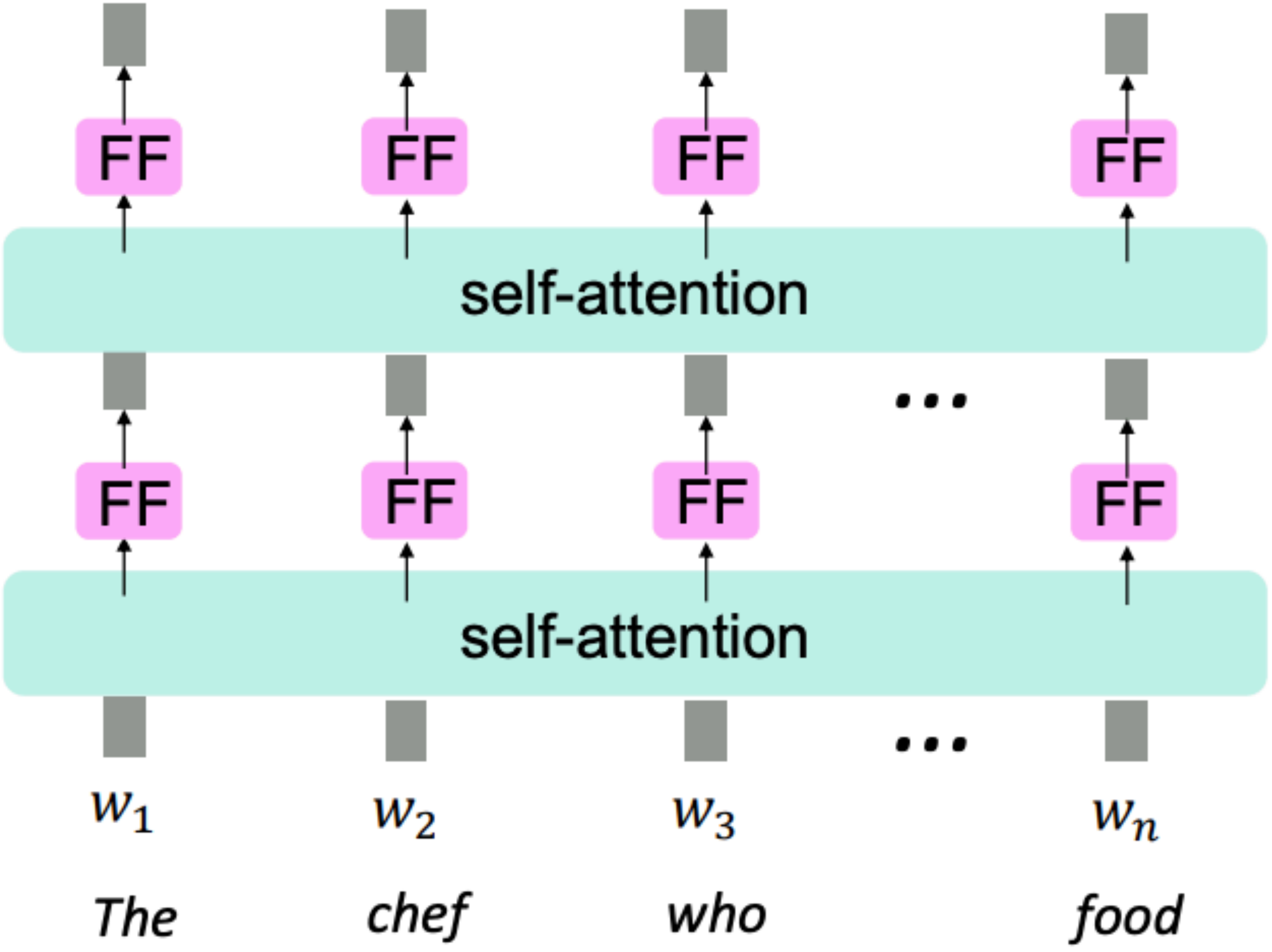
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Self-Attention and Weighted Averages

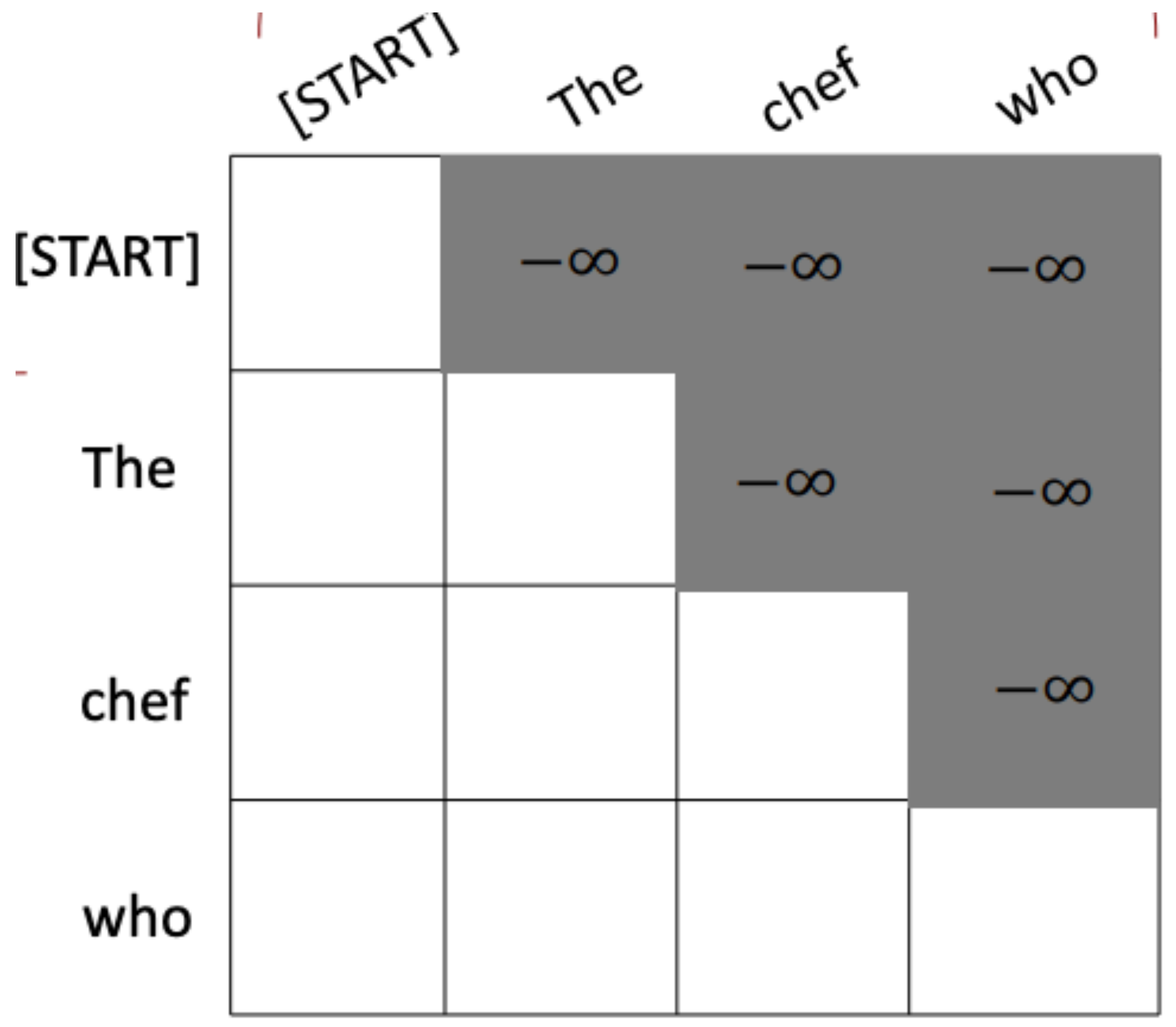
- **Problem**: there are no *element-wise* nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors
- **Solution**: add a feed-forward network to post-process each output vector.





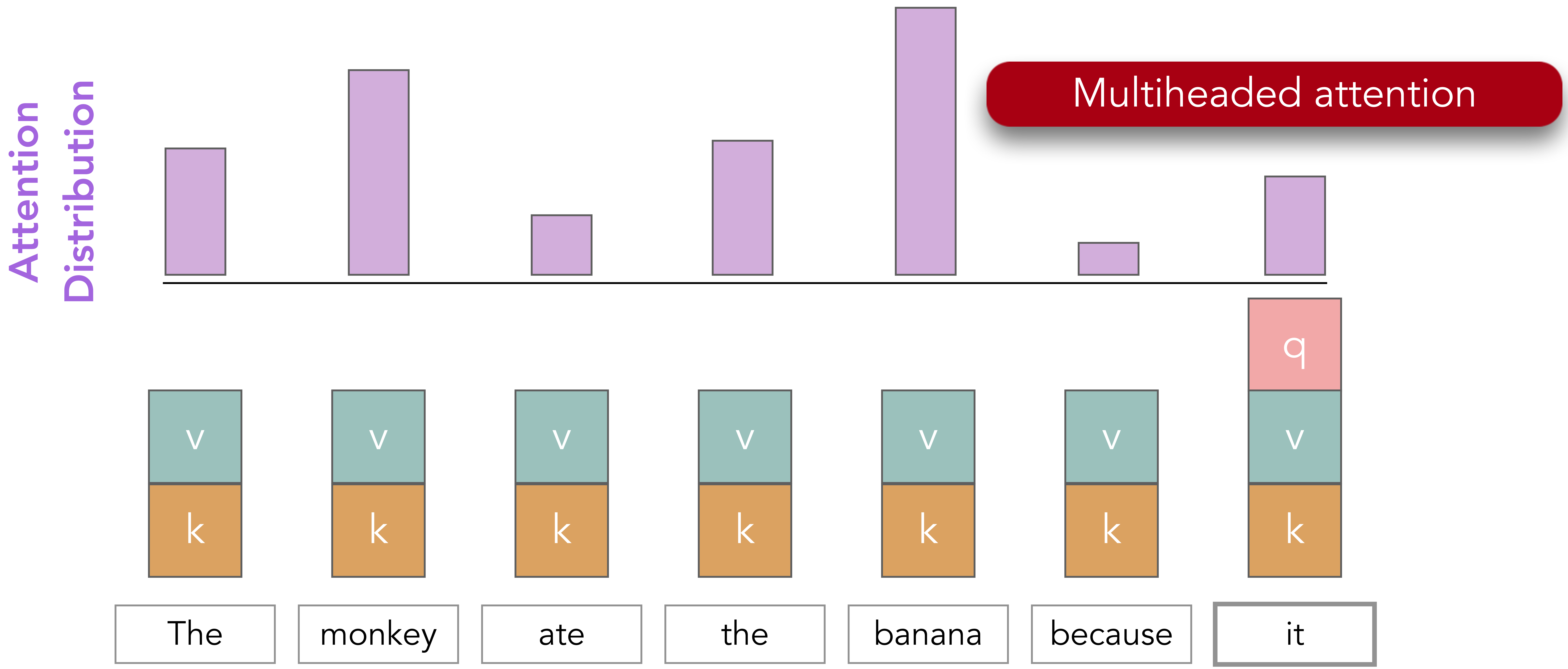
Self Attention and Future Information

- **Problem**: Need to ensure we don't "look at the future" when predicting a sequence during training
  - e.g. Target sentence in machine translation or generated sentence in language modeling
  - To use self-attention in decoders, we need to ensure we can't peek at the future.
- **Solution (Naïve)**: At every time step, we could change the set of keys and queries to include only past words.
  - (Inefficient!)
- **Solution:** To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$



Self-Attention and Heads

- What if we needed to pay attention to multiple different kinds of things e.g. entities, syntax
- **Solution**: Consider multiple attention computations in parallel

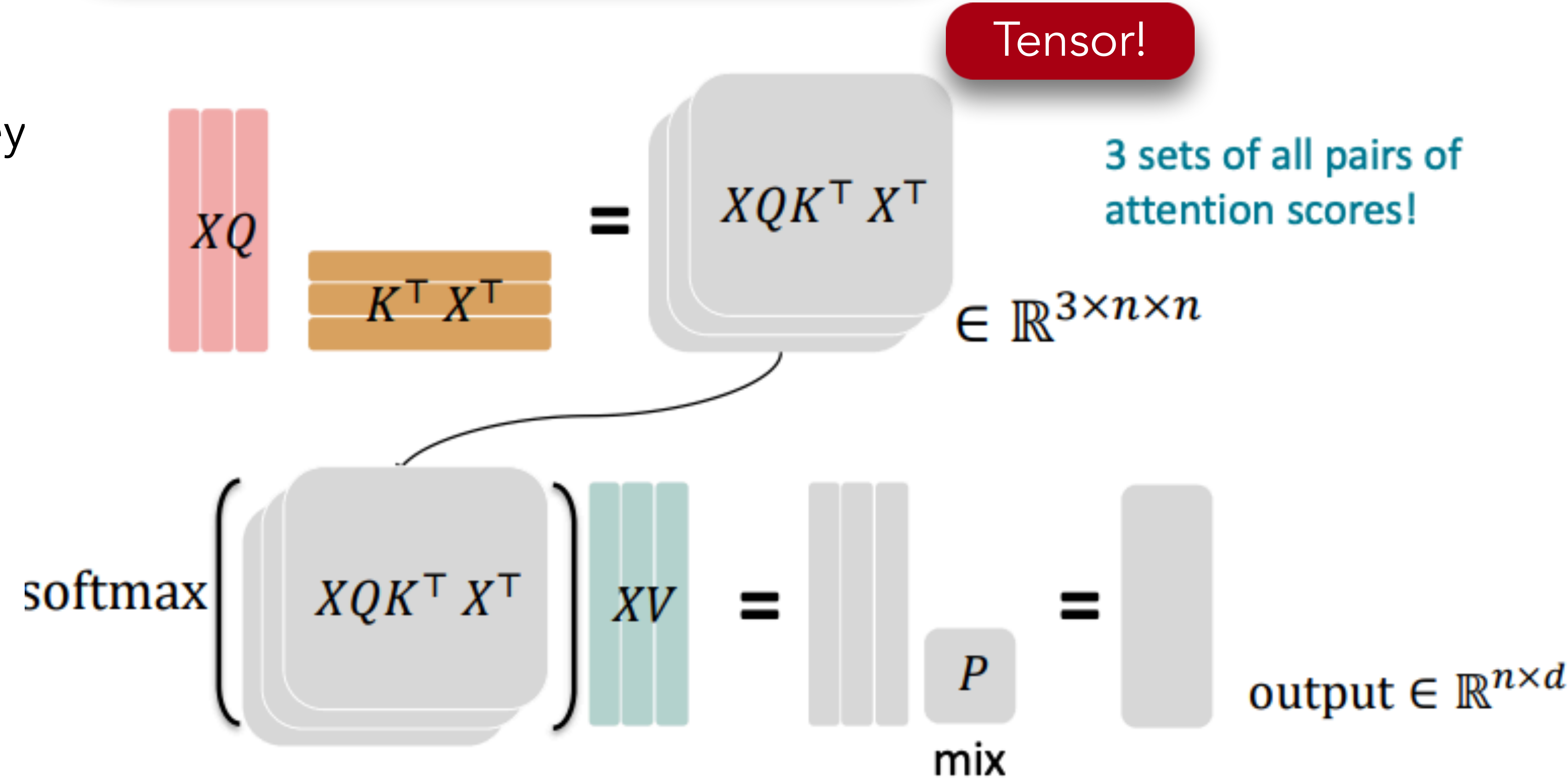


Multiheaded Attention: Visualization

Still efficient, can be parallelized!

First, take the query-key dot products in one matrix multiplication: $\mathbf{XQ}_l(\mathbf{XK}_l)^T$

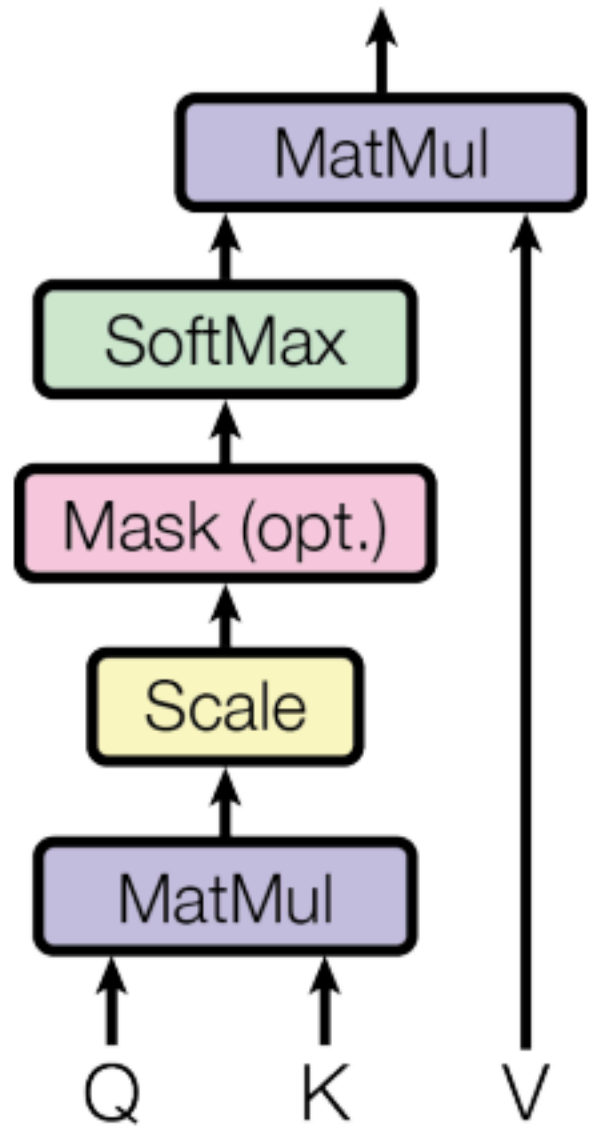
Next, softmax, and compute the weighted average with another matrix multiplication.



Scaled Dot Product Attention

$$\textbf{output}_\ell = \textbf{softmax}(XQ_\ell K_\ell^T X^T) * XV_\ell$$

- So far: Dot product self-attention
- When dimensionality d becomes large, dot products between vectors tend to become large
- Because of this, inputs to the softmax function can be large, making the gradients small
- Now: Scaled Dot product self-attention to aid in training



$$\textbf{scaled-output}_\ell = \textbf{softmax}\left(\frac{XQ_\ell K_\ell^T X^T}{\sqrt{d/h}}\right) * XV_\ell$$

- We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of d/h, where h is the number of heads

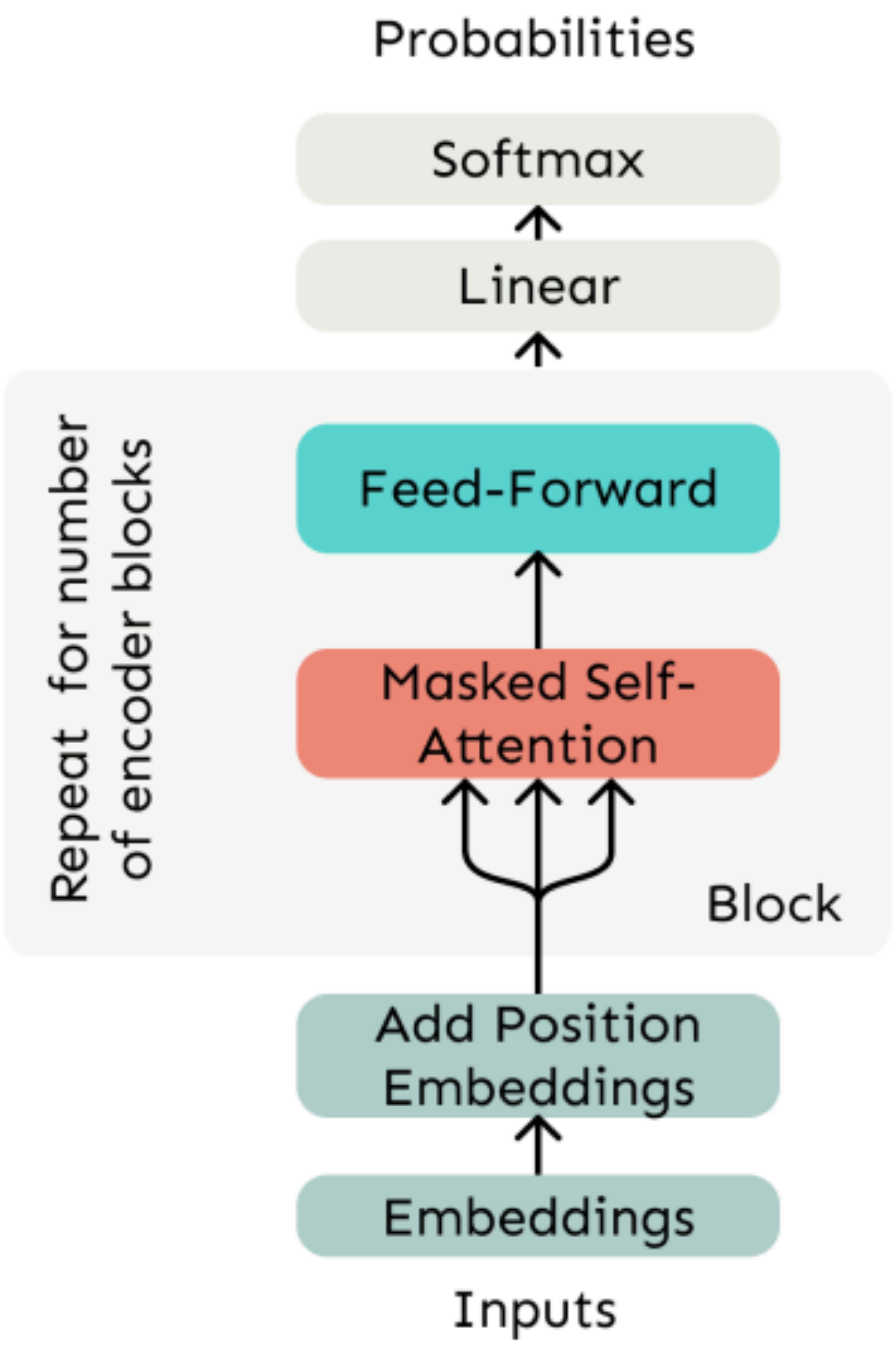
Attention is all you need (Vaswani et al., 2017)

Positional Embeddings

- Maps integer inputs (for positions) to real-valued vectors
  - one per position in the entire context
- $\mathbf{x}_i$ is the embedding of the word at index i. The positioned embedding (token embedding with position embedding) is:
  - $\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$
  - Can be randomly initialized and can let all $\mathbf{p}_i$ be learnable parameters (most common)
- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside 1, … , n, where n is the maximum length of the sequence allowed under the architecture
  - There will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits

Self-Attention Transformer Building Block

- Self-attention:
  - the basis of the method; with multiple heads
- Position representations:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- Nonlinearities:
  - At the output of the self-attention block
  - Frequently implemented as a simple feedforward network.
- Masking:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.



Lecture Outline

- Recap: Transformers
- Quiz 3
- Transformers as Encoders, Decoders and Encoder-Decoders
- The pre-training and fine-tuning paradigm
  - Pre-training Decoder-Only Models
  - Pre-training Encoder-Only Models
  - Pre-training Encoder-Decoder Models

Quiz 3

Password: recurrent

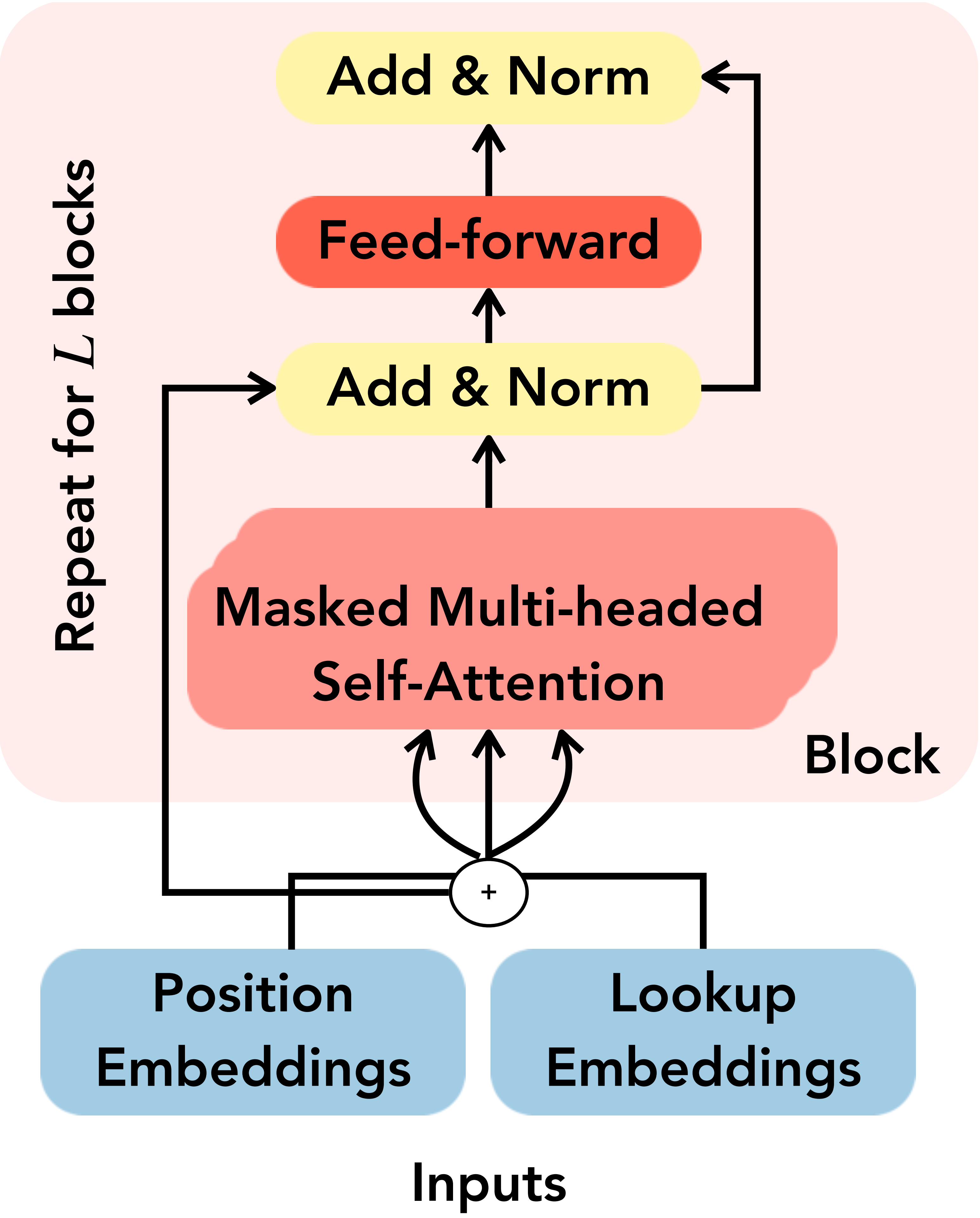
Lecture Outline

- Recap: Transformers
- Quiz 3
- Transformers as Encoders, Decoders and Encoder-Decoders
- The pre-training and fine-tuning paradigm
  - Pre-training Decoder-Only Models
  - Pre-training Encoder-Only Models
  - Pre-training Encoder-Decoder Models

Transformers as Encoders, Decoders and Encoder-Decoders

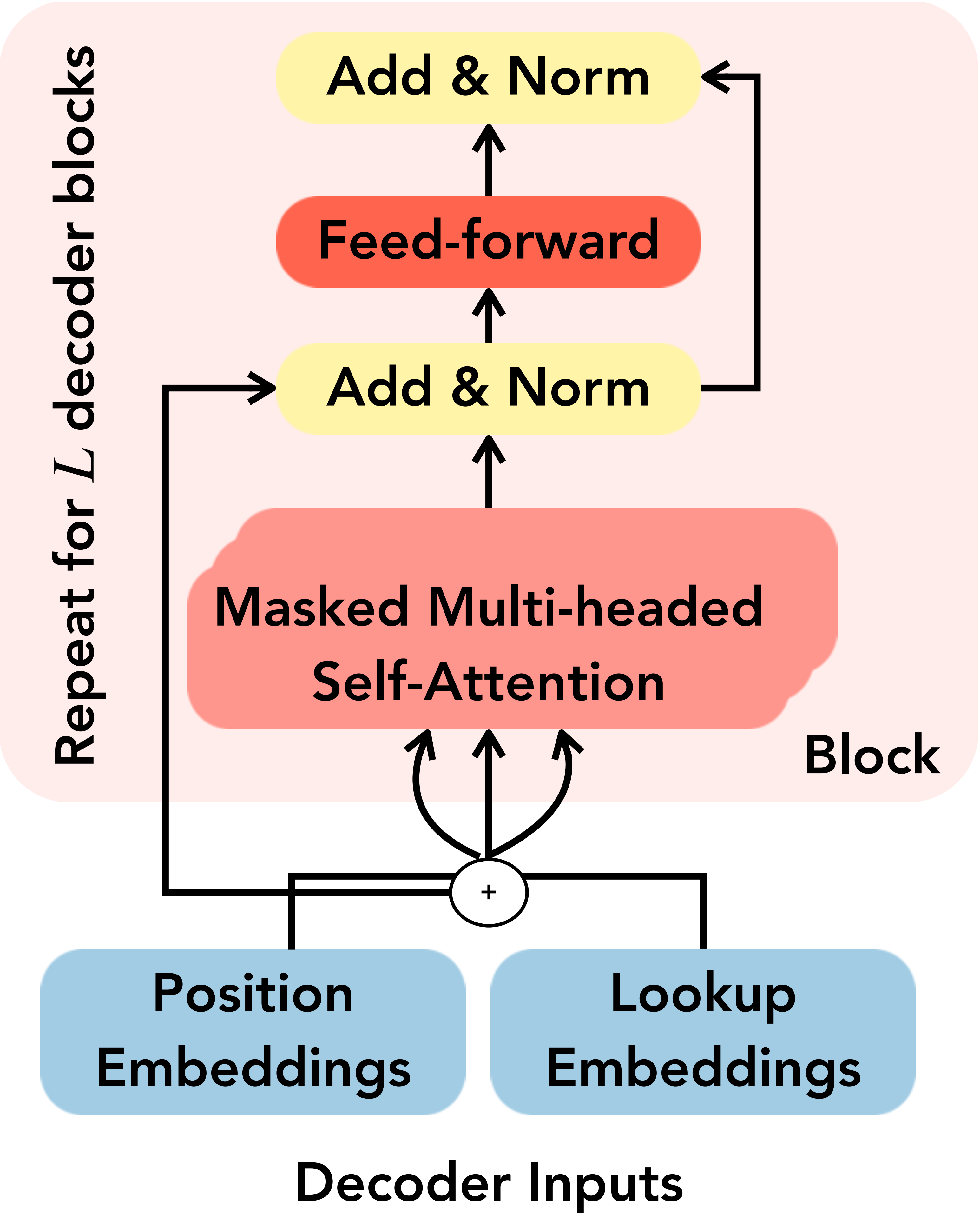
The Transformer Model

- Transformers are made up of stacks of transformer blocks, each of which is a multilayer network made by combining feedforward networks and **self-attention layers**, the key innovation of self-attention transformers
- The Transformer Decoder-only model corresponds to
  - a Transformer language model
- Lookup embeddings for tokens are usually randomly initialized
  - Input tokenization (in next lecture)



The Transformer Decoder

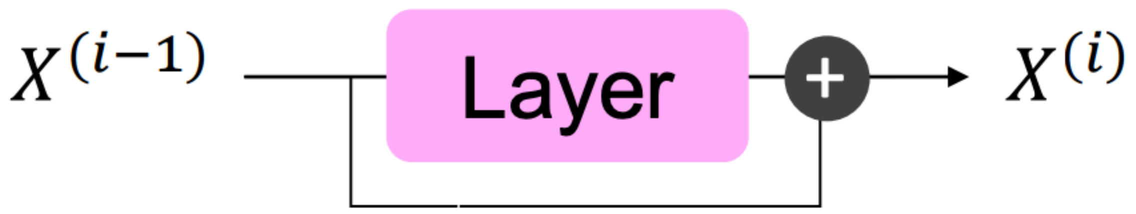
- Two optimization tricks that help training:
  - Residual Connections
  - Layer Normalization
- In most Transformer diagrams, these are often written together as “Add & Norm”
  - Add: Residual Connections
  - Norm: Layer Normalization



Transformer Decoder

Residual Connections

- Original Connections: $X^{(i)} = \text{Layer}(X^{(i-1)})$ where i represents the layer
- **Residual Connections** : trick to help models train better.
  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$
    - Helps learn "the residual" from the previous layer
    - Remember: the layer contains all the non-linearities



Allowing information to skip a layer improves learning and gives higher level layers **direct access to information** from lower layers (He et al., 2016).

Layer Normalization

- Another trick to help models train faster
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.

$$\mu = \frac{1}{d}\sum_{j=1}^{d} x_j; \quad \mu \in \mathbb{R} \qquad \sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}; \quad \sigma \in \mathbb{R}$$

Result: New vector with zero mean and a standard deviation of one

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Component-wise subtraction

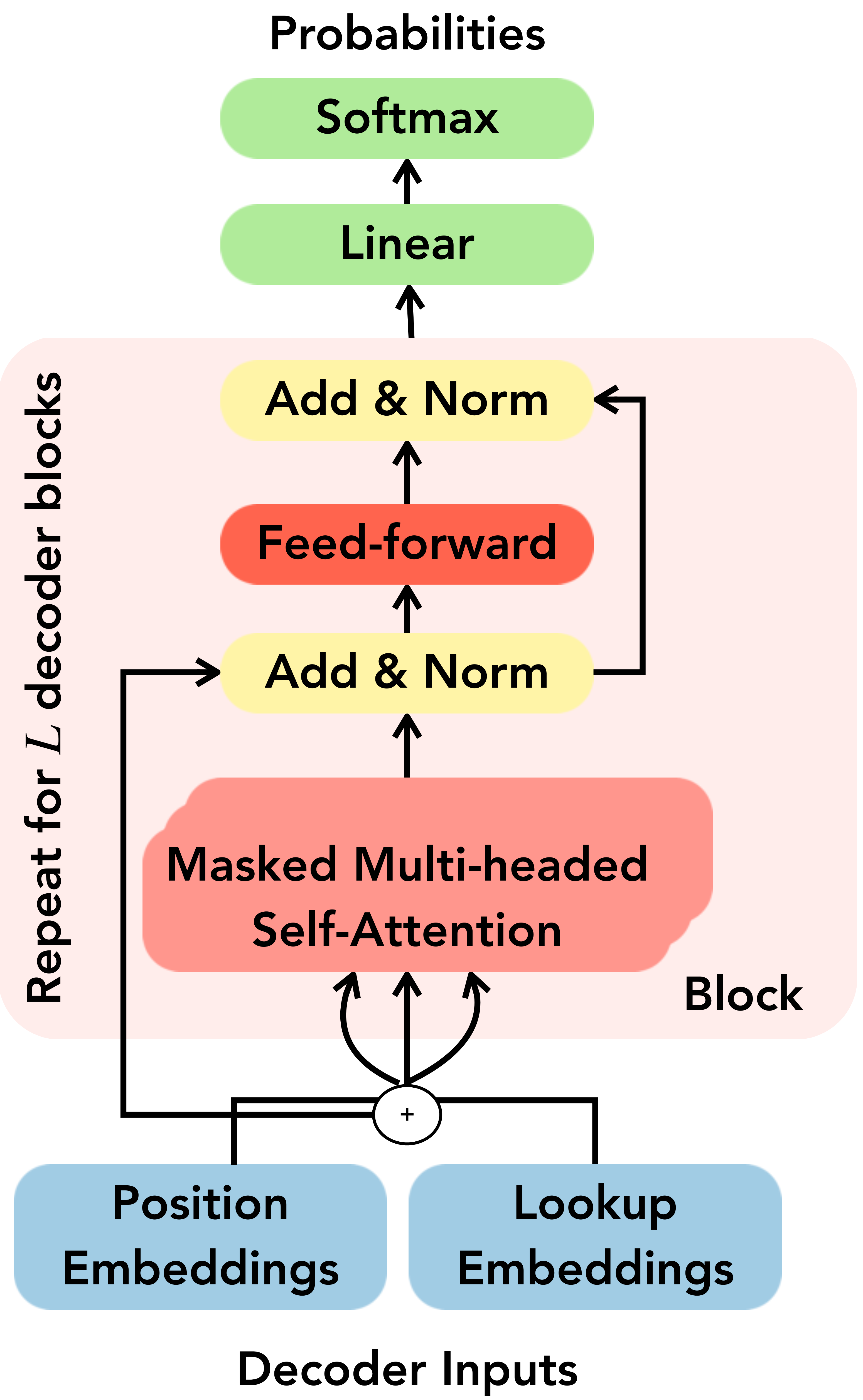
- Let $\gamma \in \mathbb{R}$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)

$$\textbf{LayerNorm} = \gamma\hat{x} + \beta$$

Xu et al., 2019

The Transformer Decoder

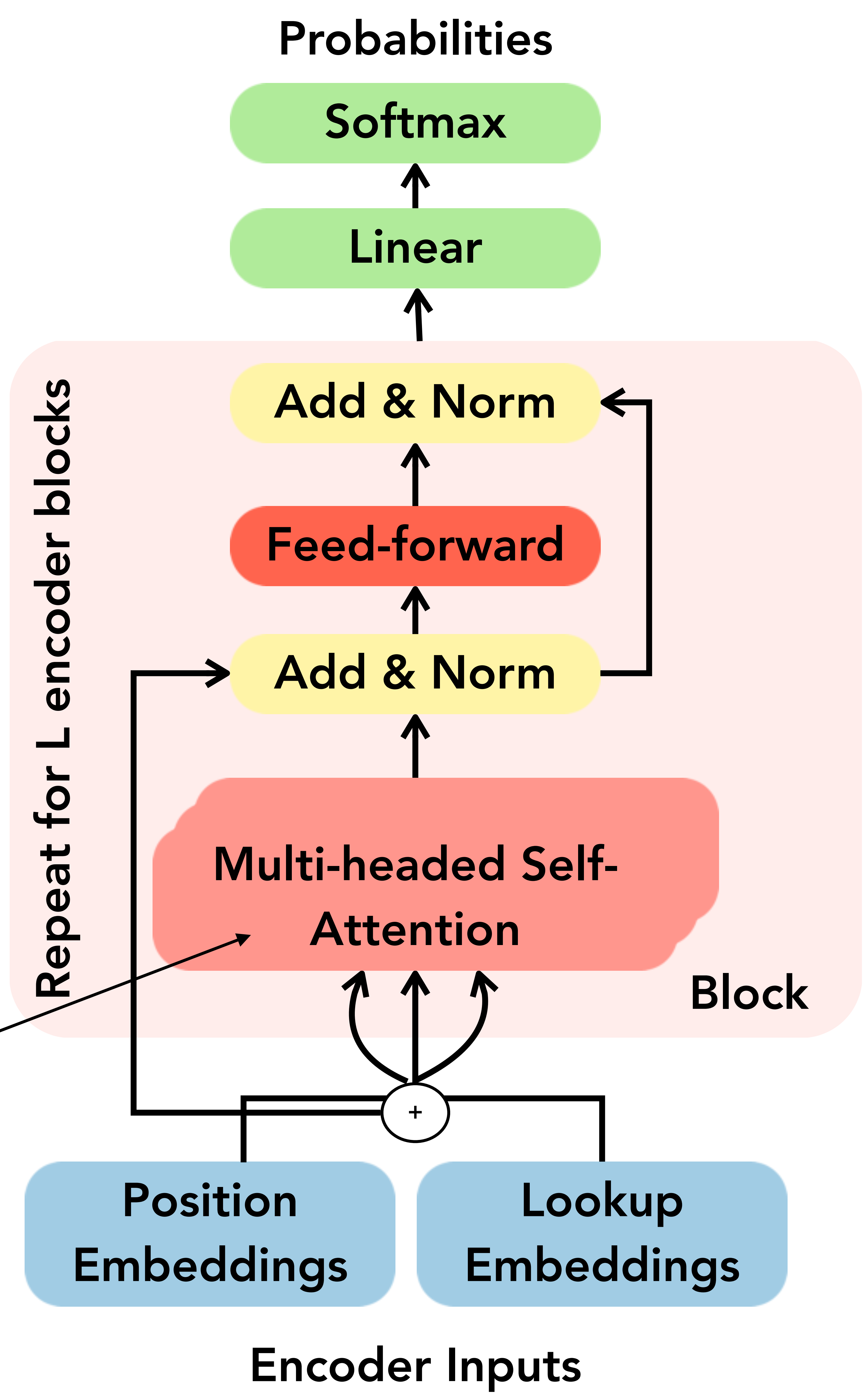
- The Transformer Decoder is a stack of Transformer Decoder Blocks.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- Output layer is as always a softmax layer



The Transformer Encoder

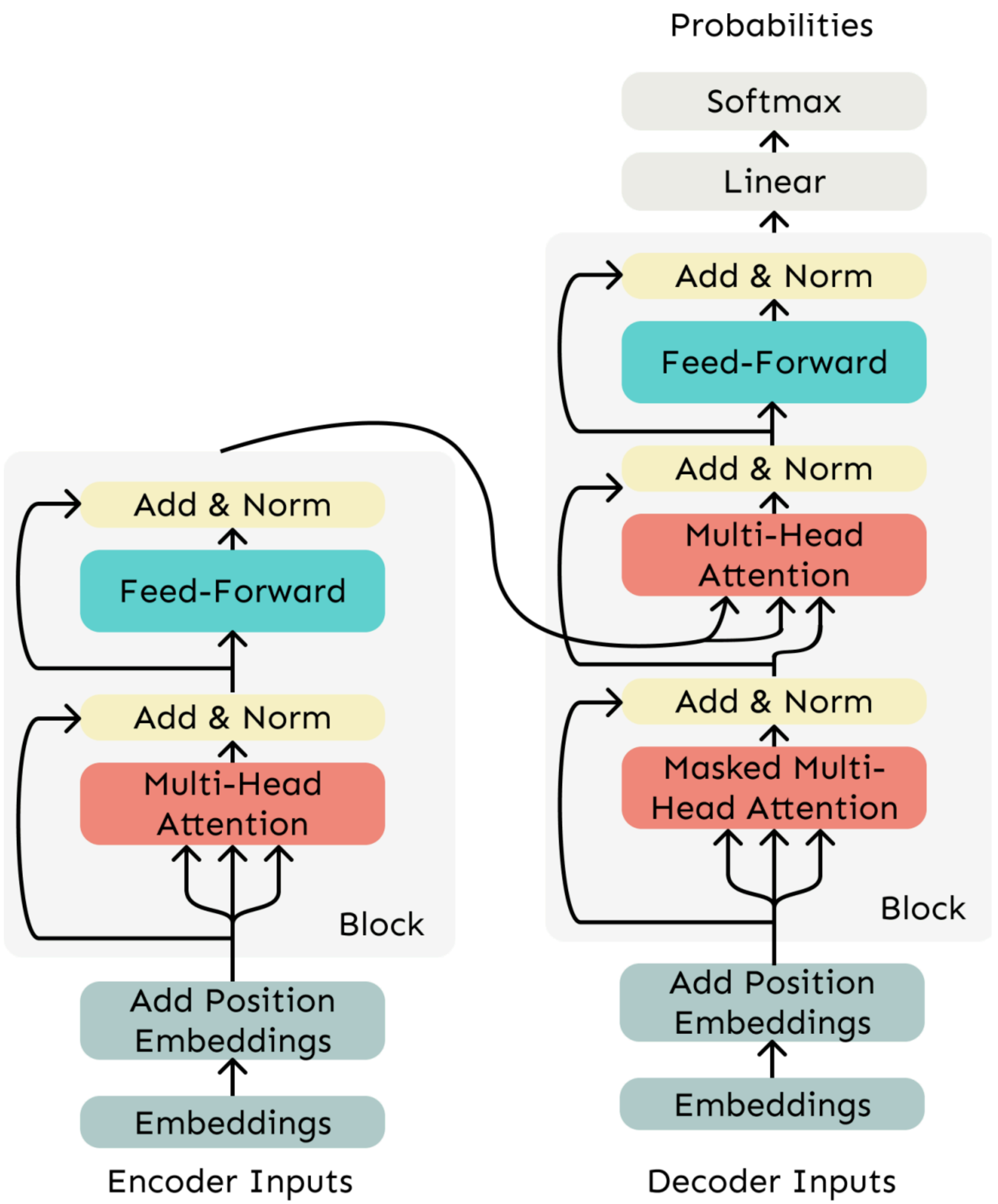
- The Transformer Decoder constrains to unidirectional context, as for language models.
- What if we want bidirectional context, i.e. both left to right as well as right to left?
- The only difference is that we remove the masking in the self-attention.
- Commonly used in sequence prediction tasks such as POS tagging
  - One output token y per input token x

No Masking!



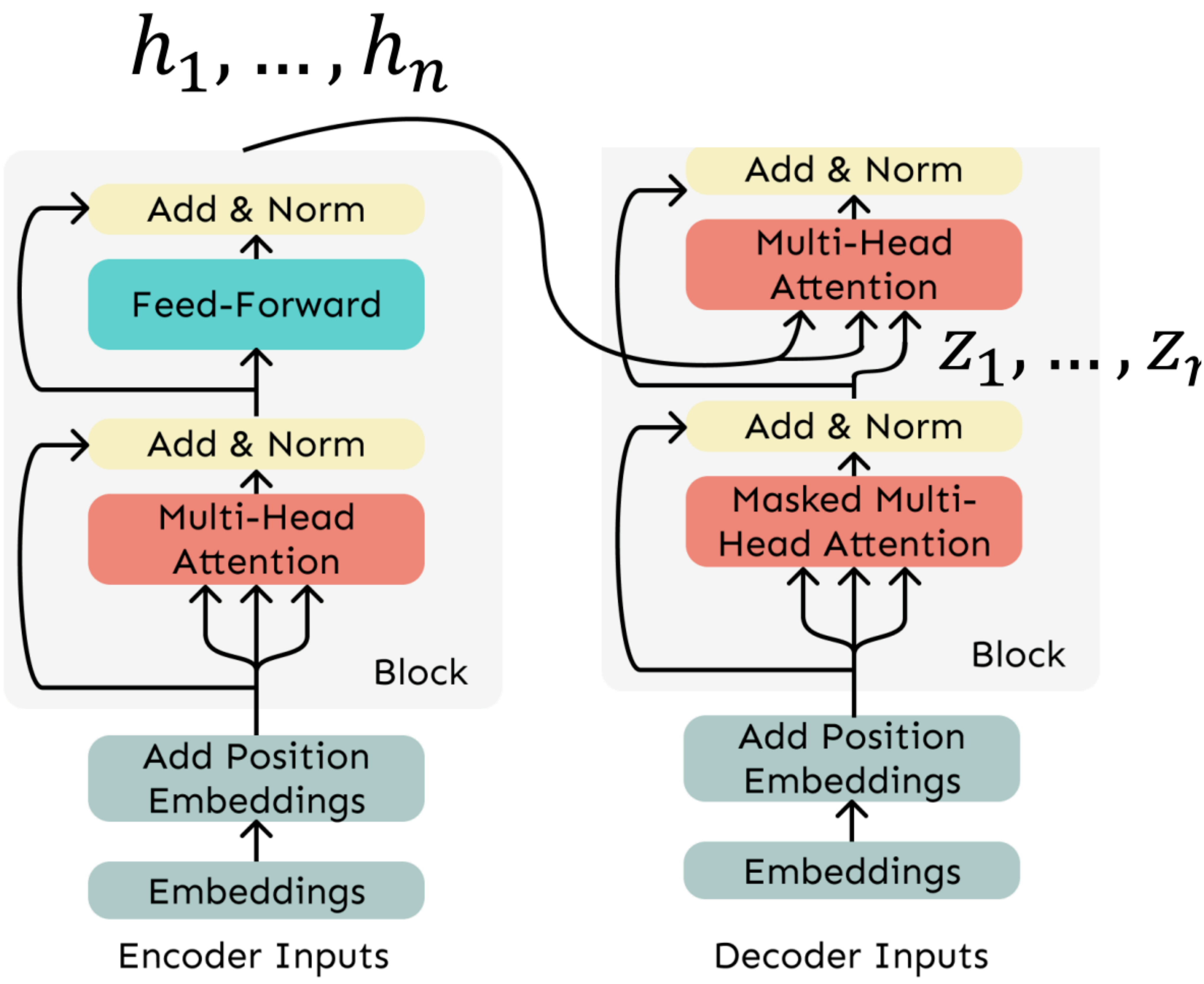
The Transformer Encoder-Decoder

- Recall that in machine translation, we processed the source sentence with a bidirectional model and generated the target with a unidirectional model.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.

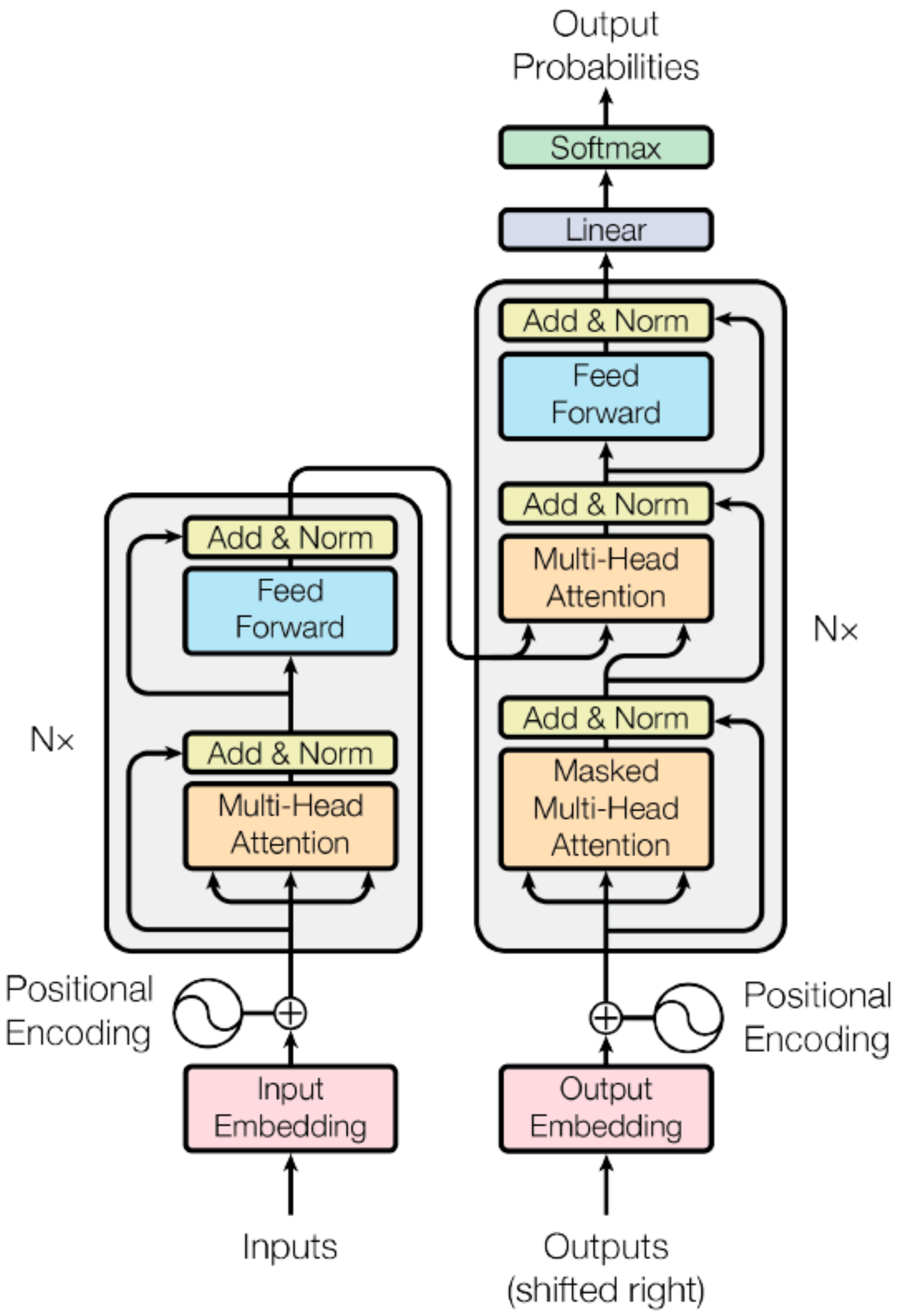


Cross Attention

- We saw that self -attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $\mathbf{h}_1, \ldots, \mathbf{h}_n$ be output vectors from the Transformer encoder; $\mathbf{h}_i \in \mathbb{R}^d$
- Let $\mathbf{z}_1, \ldots, \mathbf{z}_n$ be input vectors from the Transformer decoder, $\mathbf{h}_i \in \mathbb{R}^d$
- Then keys and values are drawn from the encoder (like a memory):
  - $\mathbf{k}_i = \mathbf{K}\mathbf{h}_i, \mathbf{v}_i = \mathbf{V}\mathbf{h}_i$
- And the queries are drawn from the decoder, $\mathbf{q}_i = \mathbf{Q}\mathbf{z}_i$



Transformer Diagram



Attention is all you need (Vaswani et al., 2017)

Transformers: Performance

Machine Translation

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Language Modeling

Model	**Test perplexity**	**ROUGE-L**
seq2seq-attention, $L = 500$	5.04952	12.7
Transformer-ED, $L = 500$	2.46645	34.2
Transformer-D, $L = 4000$	2.22216	33.6
Transformer-DMCA, no MoE-layer, $L = 11000$	2.05159	36.2
Transformer-DMCA, MoE-128, $L = 11000$	1.92871	37.9
Transformer-DMCA, MoE-256, $L = 7500$	1.90325	38.8

The real power of Transformers comes from pretraining language models which are then adapted for different tasks

Lecture Outline

- Recap: Transformers
- Quiz 3
- Transformers as Encoders, Decoders and Encoder-Decoders
- The pre-training and fine-tuning paradigm
  - Pre-training Decoder-Only Models
  - Pre-training Encoder-Only Models
  - Pre-training Encoder-Decoder Models

The Pre-training and Fine-tuning Paradigm

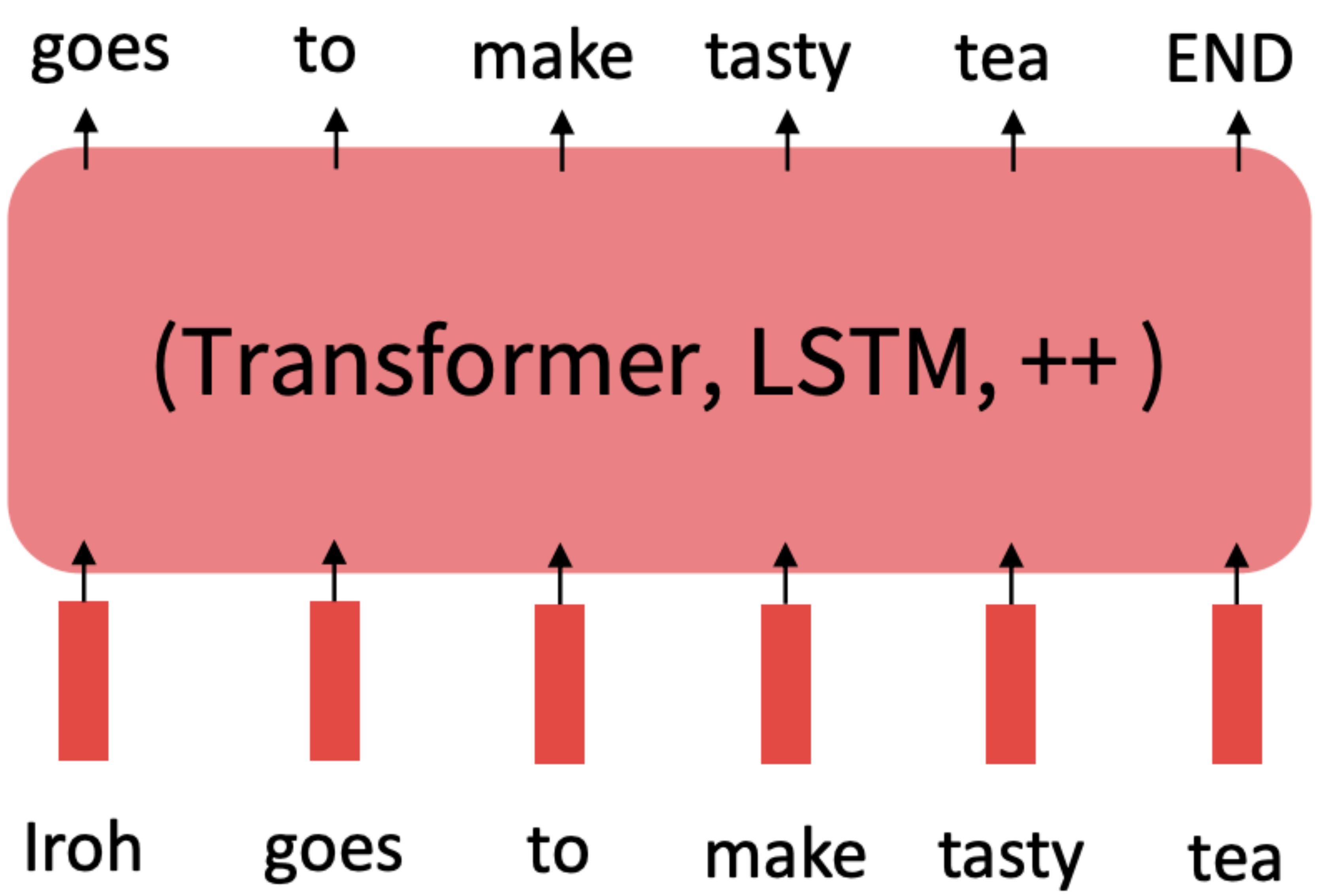
The Pretraining / Finetuning Paradigm

- Pretraining can improve NLP applications by serving as parameter initialization.

Key idea: "Pretrain once, finetune many times."

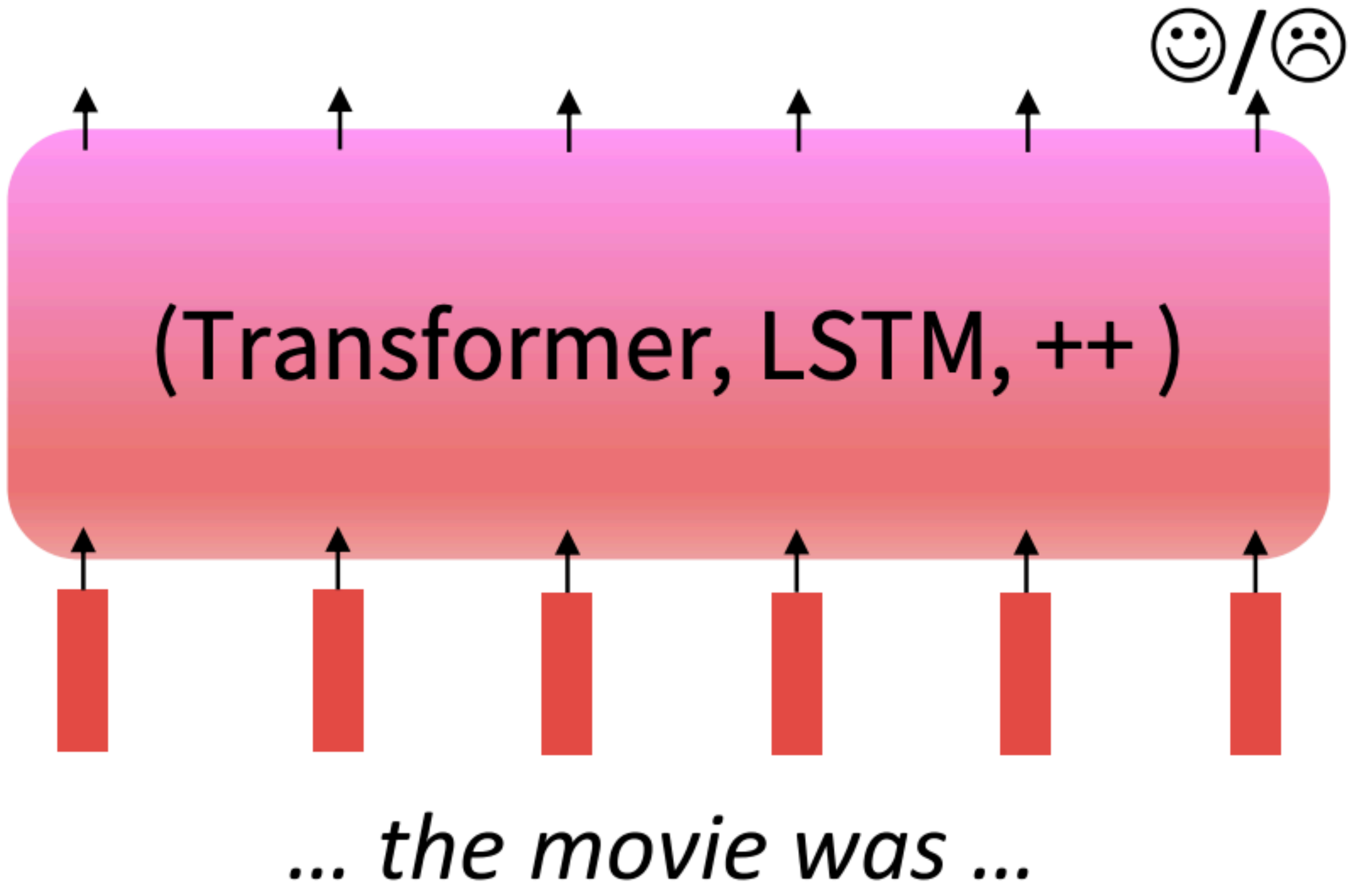
Step 1: Pretrain (on language corpora)

Lots of text; learn general things!



Step 2: Finetune (on your task data)

Not many labels; adapt to the task!

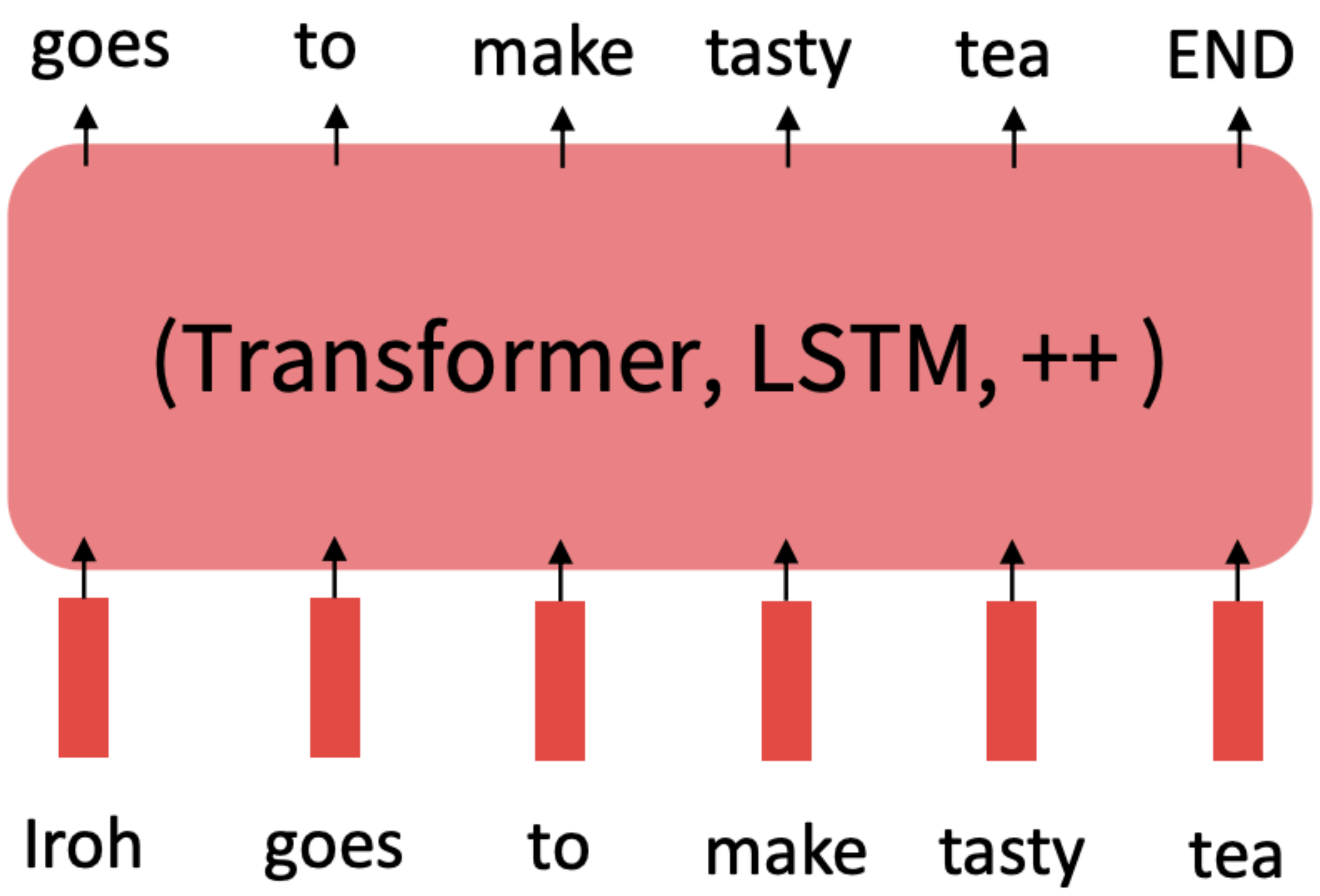


Pretraining

- Central Approach: Pretraining methods hide parts of the input from the model, and train the model to reconstruct those parts.
- Used for parameter initialization
  - Part of network
  - Full network
- Abstracts away from the task of "learning the language"

Step 1: Pretrain (on language corpora)

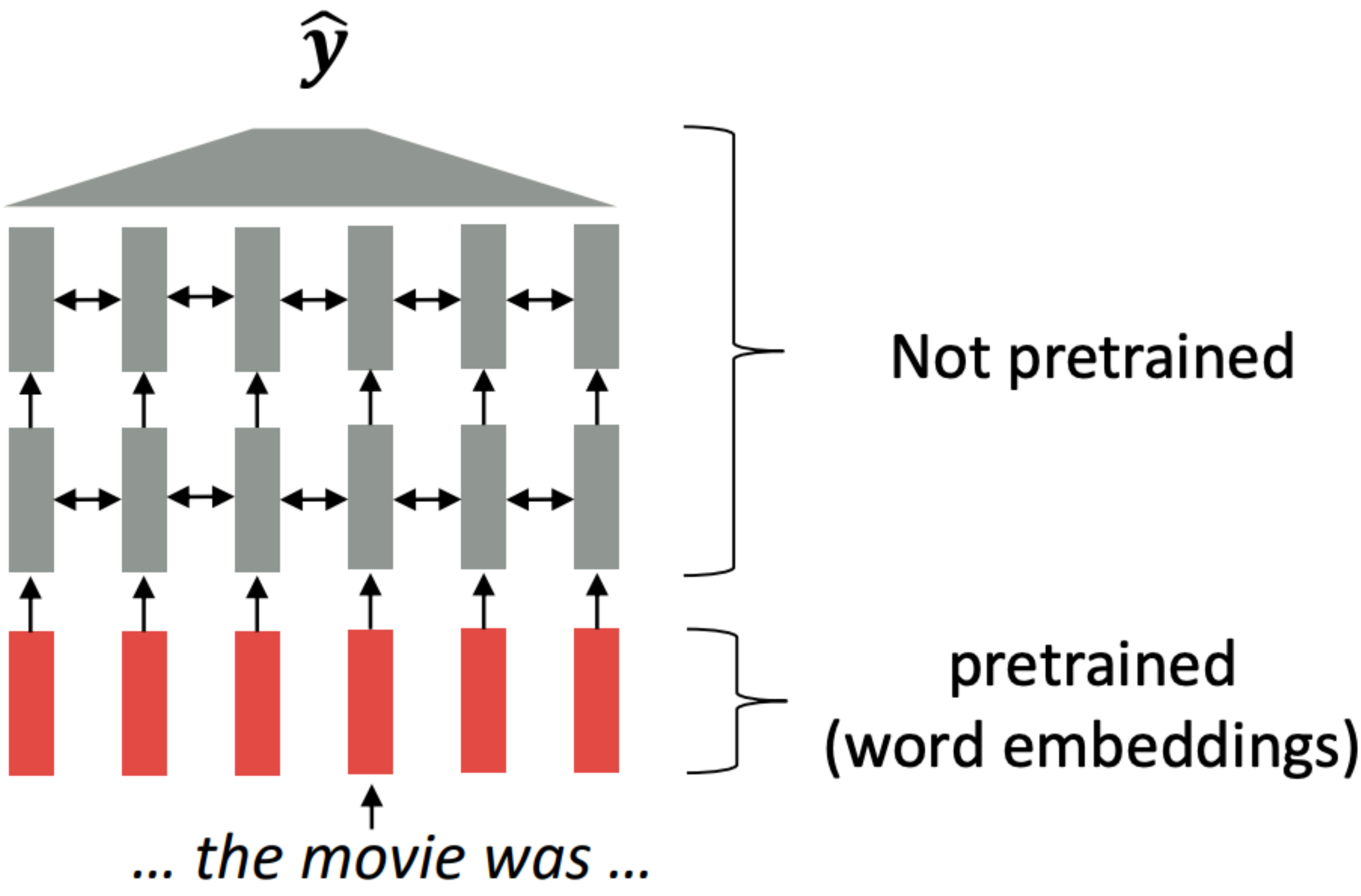
Lots of text; learn general things!



Word embeddings were pretrained too!

Previously:

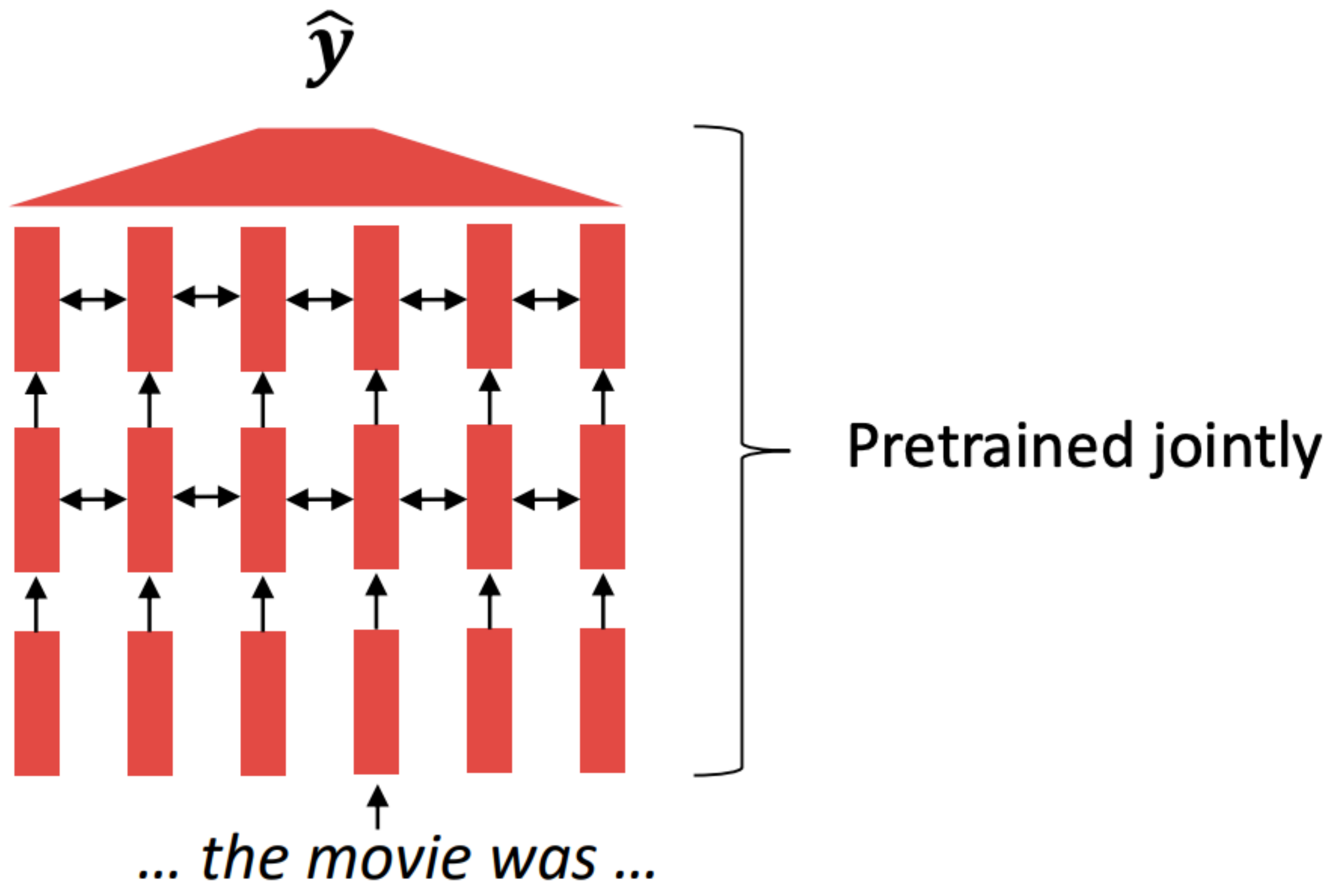
- Start with pretrained word embeddings
  - word2vec
  - GloVe
  - Trained with limited context (windows)
- Learn how to incorporate context in an LSTM or Transformer while training on the task (e.g. sentiment classification)
- Paradigm till 2017



However, the word "movie" gets the same word embedding, no matter what sentence it shows up in!

Pretraining Entire Models

- In modern NLP:
  - All (or almost all) parameters in NLP networks are initialized via pretraining.
  - This has been exceptionally effective at building strong:
    - representations of language
    - parameter initializations for strong NLP models.
    - probability distributions over language that we can sample from



[This model has learned how to represent entire sentences through pretraining]

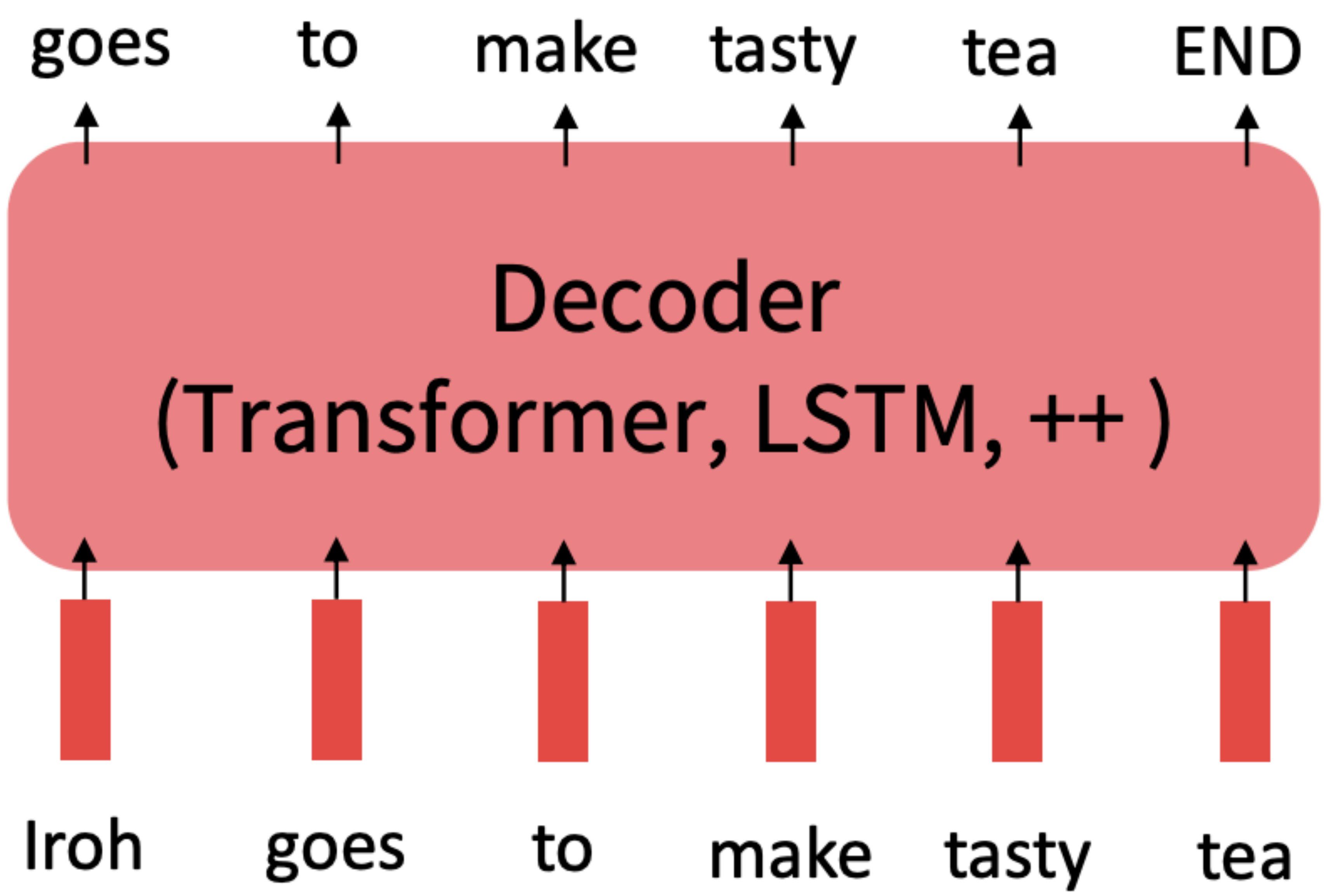
Pretraining: Intuition from SGD

Why should pretraining and finetuning help, from a "training neural nets" perspective?

- Pretraining provides parameters $\hat{\theta}$ by approximating $\min_{\theta} \mathscr{L}_{\text{pretrain}}(\theta)$
    - $\mathscr{L}_{\text{pretrain}}(\theta)$ is the pretraining loss
- Then, finetuning approximates $\min_{\theta} \mathscr{L}_{\text{finetune}}(\theta)$, **but starting at** $\hat{\theta}$.
    - $\mathscr{L}_{\text{finetune}}(\theta)$ is the finetuning loss
- The pretraining may matter because stochastic gradient descent sticks (relatively) close to $\hat{\theta}$ during finetuning
    - It is possible that the finetuning local minima near $\hat{\theta}$ tends to generalize well!
    - And/or, maybe the gradients of finetuning loss near $\hat{\theta}$ propagate nicely!

Pretraining: Language Models

- Recall the language modeling task:
  - Model $p_\theta(w_t | w_{1:t-1})$, the probability distribution over words given their past contexts.
  - There's lots of data for this! (In English.)
- Pretraining through language modeling:
  - Train a neural network to perform language modeling on a large amount of text.
  - Save the network parameters.

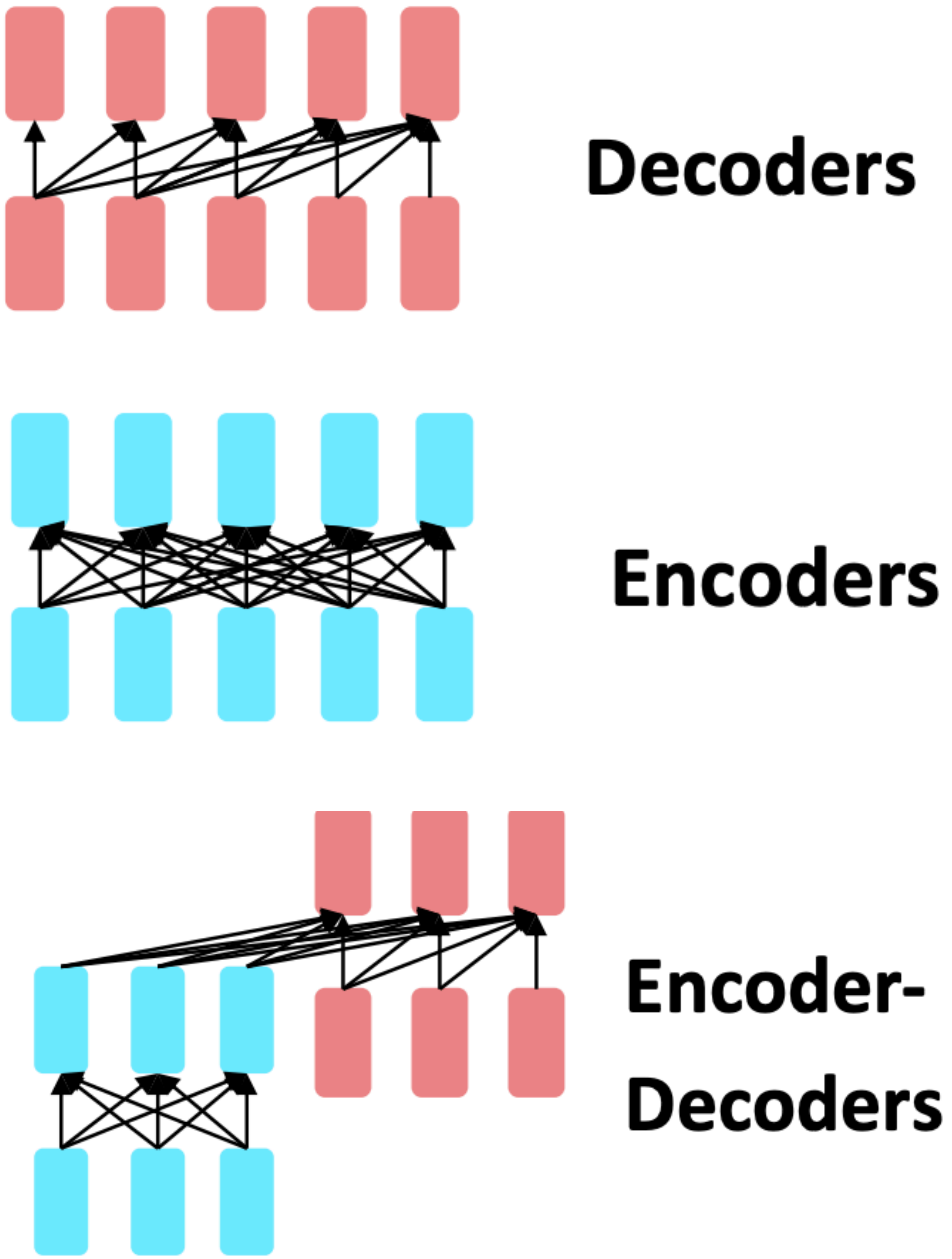


Semi-supervised Sequence Learning

Andrew M. Dai
Google Inc.
adai@google.com

Quoc V. Le
Google Inc.
qvl@google.com

Pretraining



- Not restricted to language modeling! Can be any task
- But most successful if the task definition is very general. Hence, language modeling is a great pretraining option
- Three options!